Lecture 8:
Structures, alignment, floats
Computer Architecture and
Systems Programming
(252-0061-00)
Timothy Roscoe
Herbstsemester 2012

Last time

• Procedures (x86-64): Optimizations
  – No base/frame pointer
  – Passing arguments to functions through registers (if possible)
  – Sometimes: Writing into the “red zone” (below stack pointer)
  – Sometimes: Function call using jmp (instead of call)
  – Reason: Performance
    • use stack as little as possible
    • while obeying rules (e.g., caller/callee save registers)

Dynamic Nested Arrays

• Strength
  – Can create matrix of any size

• Programming
  – Must do index computation explicitly

• Performance
  – Accessing single element costly
  – Must do multiplication

Dynamic Array Multiplication

• Per iteration:
  – Multiplies: 3
    • 2 for subscripts
    • 1 for data
  – Adds: 4
    • 2 for array indexing
    • 1 for loop index
    • 1 for data

```c
/* Compute element i,k of variable matrix product */
int * var_prod_ele
     (int *a, int *b, int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
       result += a[i*n+j] * b[j*n+k];
    return result;
}
```
Optimizing Dynamic Array Multiplication

- Optimizations
  - Performed when set optimization level to -O2
- Code Motion
  - Expression i*n can be computed outside loop
- Strength Reduction
  - Incrementing j has effect of incrementing j*n+k by n
- Operations count
  - 4 adds, 1 mult

```c
{ int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result += a[i*n+j] * b[j*n+k];
  return result;
}
```

```c
{ int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result += a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

4 adds, 3 mults

Structures

- Concept
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types
- Accessing Structure Member

```c
void set_i(struct rec *r, int val)
{
  r->i = val;
}
```

IA32 Assembly

```
# %edx = r
movl (%edx),%ecx # r->i
leal 0(,%ecx,4),%eax # 4*(r->i)
leal 4(%eax,%edx),%eax # r+4*idx
movl %eax,16(%edx) # Update r->p
```

Generating Pointer to Structure Member

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

```c
int *find_a
(struct rec *r, int idx)
{
  return &r->a[idx];
}
```

Structures Referencing (Cont.)

- C Code
  ```c
  struct rec {
    int i;
    int a[3];
    int *p;
  };
  ```
  ```c
  void set_p(struct rec *r)
  {
    r->p = &r->a[r->i];
  }
  ```

IA32 Assembly

```
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Today

- Structures
- Alignment
- Unions
- Floating point
Alignment

• **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on IA32
    • treated differently by IA32 Linux, x86-64 Linux, and Windows!

• **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    • Inefficient to load or store datum that spans quad word boundaries
    • Virtual memory very tricky when datum spans 2 pages

• **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

---

Specific Cases of Alignment (IA32)

• **1 byte: char, ...**
  - no restrictions on address

• **2 bytes: short, ...**
  - lowest 1 bit of address must be 0₂

• **4 bytes: int, float, char *, ...**
  - lowest 2 bits of address must be 00₂
  - i.e., treated the same as a 4-byte primitive data type

• **8 bytes: double, ...**
  - Windows (and most other OS's & instruction sets):
    • lowest 3 bits of address must be 000₂
  - Linux:
    • lowest 2 bits of address must be 00₂
  - i.e., treated the same as a 4-byte primitive data type

• **12 bytes: long double**
  - Windows, Linux:
    • lowest 2 bits of address must be 00₂
    • i.e., treated the same as a 4-byte primitive data type

---

Specific Cases of Alignment (x86-64)

• **1 byte: char, ...**
  - no restrictions on address

• **2 bytes: short, ...**
  - lowest 1 bit of address must be 0₂

• **4 bytes: int, float, char *, ...**
  - lowest 2 bits of address must be 00₂

• **8 bytes: double, char *, ...**
  - Windows & Linux:
    • lowest 3 bits of address must be 000₂

• **16 bytes: long double**
  - Linux:
    • lowest 3 bits of address must be 000₂
    • i.e., treated the same as a 8-byte primitive data type

---

Satisfying Alignment with Structures

• **Within structure:**
  - Must satisfy element’s alignment requirement

• **Overall structure placement**
  - Each structure has alignment requirement K
    • K = Largest alignment of any element
  - Initial address & structure length must be multiples of K

• **Example (under Windows or x86-64):**
  - K = 8, due to double element

---

Different Alignment Conventions

• **x86-64 or IA32 Windows:**
  - K = 8, due to double element

• **IA32 Linux**
  - K = 4; double treated like a 4-byte data type

---

Saving Space

• **Put large data types first**

---

Effect (example x86-64, both have K=8)

---

Structure S1 (x86-64, x64)

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Arrays of Structures

- Satisfy alignment requirement for every element

```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```

Accessing Array Elements

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
  - Resolved during linking

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

Today

- Structures
- Alignment
- Unions
- Floating point

Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

Using Union to Access Bit Patterns

```
typeunion {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}
```

```unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
} Same as (unsigned) f ?```
Summary

- Arrays in C
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system

Today

- Structures
- Alignment
- Unions
- Floating point
  - x87 (available with IA32, becoming obsolete)
  - SSE3 (available with x86-64)

IA32 Floating Point (x87)

- History
  - 8086: first computer to implement IEEE FP
    * separate 8087 FPU (floating point unit)
  - 486: merged FPU and Integer Unit onto one chip
    * Becoming obsolete with x86-64
- Summary
  - Hardware to add, multiply, and divide
  - Floating point data registers
  - Various control & status registers
- Floating Point Formats
  - single precision (C `float`): 32 bits
  - double precision (C `double`): 64 bits
  - extended precision (C `long double`): 80 bits

FPU Data Register Stack (x87)

- FPU register format (80 bit extended precision)
- FPU registers
  - 8 registers `%st(0) - %st(7)`
  - Logically form stack
  - Top: `%st(0)`
  - Bottom disappears (drops out) after too many pushes

FPU instructions (x87)

- Large number of floating point instructions and formats
  - ~50 basic instruction types
  - load, store, add, multiply
  - sin, cos, tan, arctan, and log
    * Often slower than math lib
- Sample instructions:
  - `fldz`: push 0.0
  - `flds`: load single precision real
  - `faddp`: add and pop

FP Code Example (x87)

- Compute inner product of two vectors
  - Single precision arithmetic
  - Common computation
  ```c
  float ipf (float x[], float y[], int n)
  {
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
      result += x[i]*y[i];
    return result;
  }
  ```
  ```assembly
  pushl %ebp              # setup
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%ebx       # %ebx=&x
  movl 12(%ebp),%ecx      # %ecx=&y
  movl 16(%ebp),%edx      # %edx=n
  fldz                    # push +0.0
  xorl %eax,%eax          # i=0
  cmpl %edx,%eax          # if i<n repeat
  jge .L3
  .L5:
  flds (%ebx,%eax,4)      # push x[i]
  emuls (%ecx,%eax,4)    # st(0) = x[i]*y[i]
  faddp
  incl %eax
  cmpl %edx,%eax
  jle .L5
  .L3:
  movl -4(%ebp),%ebx      # finish
  movl %ebp, %esp
  popl %ebp
  ret
  ```
  # `result = result36`
Inner Product Stack Trace

Initialization
1. \texttt{fldz} \hspace{1cm} 0.0 \hspace{1cm} %st(0)

Iteration 0
2. \texttt{flds (%ebx,%eax,4)} \hspace{1cm} 0.0 \hspace{1cm} %st(1) \hspace{1cm} %st(0)
3. \texttt{fmuls (%ecx,%eax,4)} \hspace{1cm} 0.0 \hspace{1cm} %st(1) \hspace{1cm} %st(0)
4. \texttt{faddp} \hspace{1cm} 0.0+\texttt{x[0]*y[0]} \hspace{1cm} %st(0)

Iteration 1
5. \texttt{flds (%ebx,%eax,4)} \hspace{1cm} x[0]*y[0] \hspace{1cm} %st(1)
6. \texttt{fmuls (%ecx,%eax,4)} \hspace{1cm} x[0]*y[0] \hspace{1cm} %st(1)
7. \texttt{faddp} \hspace{1cm} (x[0]*y[0]+x[1]*y[1]) \hspace{1cm} %st(0)

Today

- Structures
- Alignment
- Unions
- Floating point
  - x87 (available with IA32, becoming obsolete)
  - SSE3 (available with x86-64)

Vector Instructions: SSE Family

- SIMD (single-instruction, multiple data) vector instructions
  - New data types, registers, operations
  - Parallel operation on small (length 2-8) vectors of integers or floats
  - Example:
    \begin{align*}
    \text{Red} & \quad \times \quad \text{Blue} \quad \Rightarrow \quad \text{“4-way”}
    \end{align*}

- Floating point vector instructions
  - Available with Intel's SSE (streaming SIMD extensions) family
  - SSE starting with Pentium III: 4-way single precision
  - SSE2 starting with Pentium 4: 2-way double precision
  - \textbf{All x86-64 have SSE3 (superset of SSE2, SSE)}

SSE3 Registers

- All caller saved
- \%xmm0 for floating point return value

128 bit = 2 doubles = 4 singles

Intel Architectures (Floating Point)

<table>
<thead>
<tr>
<th>Processors</th>
<th>Architectures</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>x86-16</td>
<td></td>
</tr>
<tr>
<td>286</td>
<td>x86-32</td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>Pentium MMX</td>
<td>MMX</td>
</tr>
<tr>
<td>486</td>
<td>Pentium III</td>
<td>SSE</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>SSE2</td>
<td></td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>SSE3</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>x86-64 / em64t</td>
<td></td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>SSE4</td>
<td></td>
</tr>
</tbody>
</table>

Our focus: SSE3 used for scalar (non-vector) floating point

SSE3 Registers

- Different data types and associated instructions
  - Integer vectors:
    - 16-way byte
    - 8-way 2 bytes
    - 4-way 4 bytes
  - Floating point vectors:
    - 4-way single
    - 2-way double
  - Floating point scalars:
    - single
    - double
**SSE3 Instructions: Examples**

- **Single precision 4-way vector add:** `addps %xmm0 %xmm1`

- **Single precision scalar add:** `addss %xmm0 %xmm1`

**SSE3 Instruction Names**

- `addps`: packed (vector)
- `addss`: single slot (scalar)
- `addpd`: single precision
- `addsd`: double precision

**SSE3 Basic Instructions**

- **Moves:**
  - Single: `movss`, `movsd`
  - Double: `movdword ptr [esp+8], %eax`, `movdword ptr [esp+16], %eax`
  - Effect: `D ← S`
  - Usual operand form: reg → reg, reg → mem, mem → reg

- **Arithmetic:**
  - Single: `addss`, `subss`, `mulss`, `divss`, `maxss`, `minss`, `sqrtss`
  - Double: `addsd`, `subsd`, `mulsd`, `divsd`, `maxsd`, `minsd`, `sqrtsd`

**SSE3 Conversion Instructions**

- **Conversions**
  - Same operand forms as moves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvtss2sd</td>
<td>single → double</td>
</tr>
<tr>
<td>cvtss2ss</td>
<td>double → single</td>
</tr>
<tr>
<td>cvtsi2ss</td>
<td>int → single</td>
</tr>
<tr>
<td>cvtsi2sd</td>
<td>int → double</td>
</tr>
<tr>
<td>cvtsi2ssq</td>
<td>quad int → single</td>
</tr>
<tr>
<td>cvtsi2sdq</td>
<td>quad int → double</td>
</tr>
<tr>
<td>cvtts2si</td>
<td>single → int (truncation)</td>
</tr>
<tr>
<td>cvtts2si</td>
<td>double → int (truncation)</td>
</tr>
<tr>
<td>cvtts2siq</td>
<td>single → quad int (truncation)</td>
</tr>
<tr>
<td>cvtts2siq</td>
<td>double → quad int (truncation)</td>
</tr>
</tbody>
</table>

**x86-64 FP Code Example**

- Compute inner product of two vectors
- Single precision arithmetic
- Uses SSE3 instructions

```c
float ipf(float x[], float y[], int n) {
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++) { 
        result += x[i]*y[i];
    }
    return result;
}
```

**x86-64 FP Code Example**

```c
double funct(double a, float x, double b, int i) {
    return a*x - b/i;
}
```
Constants

```c
double cel2fahr(double temp) {
    return 1.8 * temp + 32.0;
}
```

- Here: Constants in decimal format
  - compiler decision
  - hex more readable

```
# Constant declarations
.LC2:
.long 3435973837     # Low order four bytes of 1.8
.long 1073532108     # High order four bytes of 1.8
.LC4:
.long 0              # Low order four bytes of 32.0
.long 1077936128     # High order four bytes of 32.0
```

```
# Code
cel2fahr:
mulsd .LC2(%rip), %xmm0  # Multiply by 1.8
addsd .LC4(%rip), %xmm0  # Add 32.0
ret
```

Checking Constant

- Previous slide: Claim
  ```
  .LC4:
  .long 0              # Low order four bytes of 32.0
  .long 1077936128     # High order four bytes of 32.0
  ```

- Convert to hex format:
  ```
  .LC4:
  .long 0x0            # Low order four bytes of 32.0
  .long 0x40400000     # High order four bytes of 32.0
  ```

- Convert to double:
  - Remember: e = 11 exponent bits, bias = 2^{11-1} - 1 = 1023

Comments

- SSE3 floating point
  - Uses lower ½ (double) or ¼ (single) of vector
  - Finally departure from awkward x87
  - Assembly very similar to integer code

- x87 still supported
  - Even mixing with SSE3 possible
  - Not recommended

- For highest floating point performance
  - Vectorization a must (but not in this course)
  - See next slide

Vector Instructions

- Recently, gcc can autovectorize to some extent
  - `-O3` or `-ftree-vectorize`
  - No speed-up guaranteed
  - Very limited
  - icc is currently much better

- For highest performance vectorize yourself using **intrinsics**
  - Intrinsics = C interface to vector instructions

- The latest:
  - Intel AVX: 4-way double, 8-way single

Next time

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions